Using Reconfigurable Logic to Optimise GPU Memory Accesses

Ben Cope¹, Peter Y.K. Cheung¹ and Wayne Luk²

¹Department of Electrical & Electronic Engineering, Imperial College London
²Department of Computing, Imperial College London

Abstract

Memory access patterns common in video processing algorithms, which are unsuited to the GPU (Graphics Processing Unit) memory system, are identified. We develop REDA (Reconfigurable Engine for Data Access) to improve GPU performance for such access patterns, by employing reconfigurable logic for address mapping. It is shown that a sixty times reduction in number of video memory accesses can be achieved for previously unsuited access patterns, with no detriment to well suited patterns. Surprisingly, memory access locality is also improved.

1. Introduction

The memory access requirements of graphics rendering necessitate a graphics system to be optimised for high-bandwidth localised memory access. Similarities between graphics rendering and video processing often result in impressive performance for GPU processing of high definition video [1]. However, a performance bottleneck is observed for a subset of video processing memory access patterns for which sub-optimal cache behaviour occurs [1].

A modification to a graphics system is proposed which includes a reconfigurable engine for data access (REDA) as shown in Figure 1. The REDA manages memory accesses to improve GPU performance for sub-optimal memory access patterns specific to video processing. This is possible because video processing memory access patterns are often determined pre-execution. The REDA is transparent for access patterns well matched to the memory hierarchy.

The contributions of this work are: proposal of a reconfigurable engine (REDA) to optimise GPU memory accesses; a GPU system model to explore memory system optimisations; analysis of GPU performance for memory access patterns common to video processing algorithms; presentation of the performance benefits of the REDA module.

The paper is organised as follows. In Section 2 related work is discussed. The GPU memory system is explained in Section 3. Section 4 includes analysis of GPU performance. The REDA module is described in Section 5. Section 6 contains results of performance gains from including the REDA. The work is concluded in Section 7.

2. Related Work

Prior work on the optimisation of GPU memory accesses has focused on software-based techniques which balance compute power and cache bandwidth of the GPU [2, 3]. A performance improvement of at most 2–5 times is achieved through optimal cache blocking [2] or splitting a matrix-matrix multiply into smaller blocks [3]. Both require algorithm separation over multiple execution passes with increased pipeline setup and computation overheads.

The work presented here proposes a reconfigurable engine for data access to optimise cache use in the GPU. This optimisation is possible because the application domain is limited to video processing algorithms.

The following two algorithms exemplify the requirement for the REDA module. Histogram equalisation and motion vector estimation have a throughput of only 5.8 and 6.7 frames per second respectively, for 720p video frame, when implemented on a GeForce 7 GPU [1]. Both contain no or partially overlapping window operations implemented in multiple execution passes. Although the implementations utilise optimisation techniques from the literature [3], the performance is low. This work shows how these window operations, more precisely the associated memory access patterns, can be optimised with the REDA module.

3. GPU Memory System

A conceptual model of the GPU memory system is shown in Figure 1. The reconfigurable engine for data access (REDA) is also shown and is described in Section 5.

The GPU memory system comprises separate read and write clients. The clients arbitrate for video memory access through a memory channel. The memory access pattern \(A\) from GPU cache ‘read clients’ is the focus of this work. Other clients, which access the video memory, include output buffers, rasterisation and vertex processing.

The memory access pattern \(A\) output from GPU cache is determined by four factors: processing (more precisely rasterization) order \((P)\), cache scheme, GPU kernel multithreading \((T)\) and GPU kernel memory requests. The processing order \((P)\) is fed in batches \((T)\) to the GPU kernels.
For this work processing order \((P)\) is a localised z-pattern [5] over a rectangle of target frame size.

A GPU kernel is executed in multiple threads across multiple processor cores [4]. The multi-threading serves two purposes: to keep processor execution pipelines full; and to hide off-chip memory access latency. The number of kernel threads \((T)\) in flow at once is predicted to be of the order of one to two thousand [6]. Choice of GPU kernel and number of threads \((T)\) determines the memory request order \((C)\).

The GPU cache determines the video memory access pattern \((A)\) from cache requests \((C)\). Previous GeForce 6 cache estimates involve a 4-way associative 16KByte cache with \(8 \times 8\) pixel cache lines [7]. For the GeForce 7, cache size is predicted to have increased to 128KBytes [2].

A typical choice of video memory is DRAM which has an increasing trend in high latency, high bandwidth specification [8]. DRAM technology is becoming increasingly sensitive to memory access locality with increasing memory access latency [8]. DRAM burst length \(B\) equals 4 for GDDR3 DRAM [9]. Video memory and the Host are the only blocks in Figure 1 which are ‘off-chip’.

### 4. GPU Performance

Equation 1 is a frequently occurring video processing kernel. Each output pixel \(F_{out}(x, y)\) represents a thread in set \(P\) and \((x_{ref} + i, y_{ref} + j)\) an address in the set \(C\).

The size of the input and output frames \(F_{in}\) and \(F_{out}\) are \(N_{in} \times M_{in}\) and \(N_{out} \times M_{out}\) pixels respectively. The product \(N_{out} \times M_{out}\) equals \(|P|\), where || is the cardinality. Ideally one would like \(N_{in} \times M_{in}\) to equal \(|A|\). That is each input pixel is only accessed once.

\[
F_{out}(x, y) = \sum_{i=0}^{l-1} \sum_{j=0}^{J-1} F_{in}(x_{ref} + i, y_{ref} + j) \quad (1)
\]

\[
x_{ref} = (I - O_x)x, \quad y_{ref} = (J - O_y)y \quad (2)
\]

The reference location \((x_{ref}, y_{ref})\) in the input frame \(F_{in}\) is a linear function of the current output pixel location \((x, y)\) as shown in Equation 2, where \(O_x, O_y\) is the overlap in accessed portions of \(F_{in}\) for neighbouring outputs.

![Figure 1. Modified GPU memory system: illustrating processing \((P)\), cache request \((C)\) and memory access \((A)\) patterns [1, 4]](image)

![Figure 2. GPU Performance Distribution](image)

Two special cases of Equation 2 are where \(x_{ref} = x, y_{ref} = y\) (case A) and \(x_{ref} = Ix, y_{ref} = Jy\) (case B). Case A represents maximal overlap of input frame windows for neighbouring output pixels. Case B is when there is no overlap of input frame windows. Examples of case A, case B and the general case are found in a 2D convolution filter, decimation and motion vector estimation respectively. The performance of the kernel in Equation 1 is now analysed to identify potential for memory access pattern optimisation.

The test hardware is a GeForce 7900 GTX GPU with 512MB of video memory. Performance is taken as the execution time to compute one output frame. For accuracy 200 rendering passes are averaged for each result, and a \(C + +\) model is used to verify output data. Output frames are rendered to off-screen targets and input textures are in 3-vector byte format as is common for video data.

To show relative performance for changes in \(I, J, O_x\) and \(O_y\), the total number of memory requests by kernels \(|C|\) is kept constant \(|C| = IJN_{out}M_{out}\). For 2D window symmetry \(I\) and \(J\) are set equal, similarly \(O_x = O_y\).

Figure 2 shows the results of varying overlap and window size. An initial window size of \(4096 \times 4096\) pixels is chosen which is the maximum texture size of a GPU and is used to show memory access variations more prominently. Paths of the special cases A and B are highlighted.

For case A the performance is approximately constant over all window size. In contrast case B shows an increase in time per frame when window size is increased. The performance of case A and B is proportional to \(|C|\) and \(|A|\) respectively. This relationship is reasoned below.

GPU computations are grouped into \(\hat{T}\) thread batches (subsets of \(P\) of size \(\hat{T}\)) over \(n\) processors. It is reasonable to assume that each GPU kernel instruction is executed on each thread (one thread on each of \(n\) processors) in turn to hide memory access and computation latencies. This has the effect that each \((i, j)\) step of Equation 1 is executed over \(\hat{T}\) threads before the next \((i, j)\) step is executed. This is referred to as simultaneous interleaved multi-threading.

It can be assumed that threads are grouped into 2D
blocks of size $\hat{T}$. Under this condition there is a step of $I,J$ in $x$ and $y$ dimensions for neighbouring threads in case B for one value of $(i,j)$. For $(i+1,j)$ thread block requests are shifted by one location in $x$. Cache line size $(L)$ is predicted to be $8 \times 8$ pixels and cache size $128$ KBytes. It is seen that for large $\hat{T}$ cache ‘thrashing’ can occur if cache size $S$ is less than $3\hat{T}\min(I \times J, L)$. For large $\hat{T}$ (1k–2k) the equality is broken. As $I,J$ increases the cache ‘thrashing’ effect amplifies as shown for case B in Figure 2.

For case A cache ‘thrashing’ does not occur because memory requests of neighbouring threads are in consecutive memory locations. Case A is performance limited by the number of memory requests $(|C|)$, a constant in Figure 2.

For the general case of Equation 2, an interesting feature of Figure 2 occurs at the point where $I - O_x$ equals 8. This is where memory access steps are equal to the predicted dimensions $(8 \times 8)$ pixels of GPU cache lines. Considering the analysis for case B this is where each thread requests pixels from a different cache line. This is shown in Figure 2 by a peak in time taken per frame. The remainder of the surface shows a trend between the two extremes of cases A and B. The contours where the value of $I - O_x$ is constant have equal performance. This is because cache performance is approximately consistent under these conditions.

In summary, case A exhibits a memory request pattern which makes optimal use of the GPU cache. Case B involves a situation where cache use is increasingly inefficient for increased window size. The focus of optimisations by the REDA module must therefore be on non- (case B) or partially overlapping (general case) windows in Equation 1.

5. REDA Architecture

A REDA module is added to the GPU memory system as shown in Figure 1. The main blocks are address mapping, buffering and control. Mapping $R$ must be chosen to optimise GPU cache behaviour and minimise $|A|$.

Figure 3 summarises the REDA architecture. The buffering and control blocks handle data rearrangement into cache lines. Alternate mapping schemes are possible for different input textures (frames). A test case of a single texture input is assumed in this work. Further, a one-to-one address mapping with DRAM burst length granularity reordering is applied, simplifying the buffer and control block designs. The implementation detail therefore focuses on the address mapping block which is the critical path.

Two design requirements are: the module must be adaptable to different access patterns, and access pattern variations must be handled at runtime. Reconfigurable logic provides an efficient platform for supporting this flexibility.

Memory access optimisation requires two stages. First, apply an addresses mapping $(R)$ between the GPU cache and address generator. Second, to compensate the reverse mapping $(R^{-1})$ is applied to the request pattern of kernels. For data continuity the mapping $R$ must be an invertible function. A choice of $R$ to optimise poorly performing access patterns from Section 4 is presented here.

Section 5.1 describes the address mapping $R$ for case B (non-overlapping windows) and the generalisation to windows with partial overlap. An FPGA prototype of the reconfigurable logic implementation is discussed in Section 5.2.

5.1. Memory Address Mapping

The proposed mapping is to translate steps in memory accesses to consecutive memory addresses which is done with respect to the target reduction size $(I,J)$ and a threading factor $(T)$. To maintain the video memory granularity, bursts of $\sqrt{B} \times \sqrt{B}$ pixels remain in sequential order.

The REDA block mapping from a requested address $(x_{addr}, y_{addr})$ in $A$ to the memory access address $(x_{map}, y_{map})$ in $R(A)$ is defined in Equations 3 to 6 for $x_{map}$, where $mod$ is the modulo operator. The form of $y_{map}$ is identical to $x_{map}$. The value $\sqrt{T}$ is the threading factor in $x$ and $y$ dimensions. Note that threading $T$ is different to the number of GPU threads $\hat{T}$ mentioned previously.

$$x_b = x_{addr} \mod \sqrt{B} \quad (3)$$
$$x' = (x_{addr} - x_b) \mod I\sqrt{T} \quad (4)$$
$$x'' = \sqrt{B} \lceil \frac{x'}{\sqrt{B}\sqrt{T}} \rceil, \quad x''' = \lfloor x''/\sqrt{B} \rfloor \mod \sqrt{T} \quad (5)$$
$$x_{map} = I\sqrt{T} \left( \frac{x_{addr} - x_b}{I\sqrt{T}} \right) + x'' + Ix''' + x_b \quad (6)$$

The generalised mapping requires small modifications to Equations 4 and 6 as shown in Equations 7 and 8. The window dimension $I$ is substituted with $I' = (I - O_x)$. This is the predicted cache steps between threads as discussed in Section 4. The mapping for $y_{map}$ is similarly generalised.

$$x' = (x_{ref} - x_b) \mod I'\sqrt{T} \quad (7)$$
$$x_{map} = I'\sqrt{T} \left( \frac{x_{addr} - x_b}{I'\sqrt{T}} \right) + x'' + I'x''' + x_b \quad (8)$$

5.2. Reconfigurable Logic Implementation

The REDA module presents four challenges for reconfigurable logic implementation. It must be:

1. High speed to match GPU core clock frequency ranging from 430 to 650 MHz on GeForce 7 GPUs.
2. Adaptable to the different mapping schemes including the ones specified in Equations 3 to 6.
3. Low latency to minimise additional impact on the already large off-chip DRAM latency.

4. Transparent for already optimal access patterns.

Challenge four is achieved, with a small data path delay, through the choice of a customised routing of only wire interconnects under the condition of already optimal access patterns. The other challenges are addressed below.

It is assumed that an access is made up of an \((x, y)\) address and a texture identifier. These are later combined to form a memory address in the address generator in Figure 1. Each \((x, y)\) location is 12-bits long, this is sufficient to represent the maximum texture dimension of 4096 pixels.

Although the above mapping appears complex, its implementation in reconfigurable logic is efficient. The design can be simplified to bit manipulation for the special case when \(\sqrt{T}, I, J\) are powers of two and \(\sqrt{B} = 2^i, i = 0, 1, 2, \ldots\). The bit rearrangement is shown in Equation 10, with \(m = \log2(I) + \log2(\sqrt{T})\) and \(b = \log2(\sqrt{B})\).

\[
x = x_{11}x_{10}x_9x_8x_7x_6x_5x_4x_3x_2x_1x_0
\]

\[
x_{\text{map}} = x_{11} \cdots x_m x_{\log2(1)} \cdots x_b x_{m-1} \cdots x_{\log2(1)} + 1 x_b - 1 \cdots x_0
\]

If window dimensions \(I, J\) are not a power of two then the implementation is more complex. Parameter \(\sqrt{T}\) can be selected to be a power of two and \(\sqrt{B}\) is often a power of two because it matches DRAM burst length. Cases where \(\sqrt{T}\) is an integer are considered here, however, in general any value of variable \(I, J, \sqrt{T}, \sqrt{B}\) can be implemented.

Prototype mapping’s, implemented on a Xilinx Virtex 5 xc5vlx30 device, have the resource usage and latency shown in Table 1. It is clarified that the Virtex 5 is a prototype platform. The proposal is of a small embedded reconfigurable unit to be included in the GPU architecture.

The latency of the design is 1 cycle for the case in Equation 10 and 8 cycles when \(I, J\) are non power of two values. The minimum clock period, for non-power of two \(I, J\) values, is 1.422 ns (taken from post place and route results in Xilinx ISE 9.1i). This suggests that the specified maximum DSP48E speed of 550 MHz is achievable. For a small and specially designed, in-system, reconfigurable unit it is reasonable to expect that a higher clock speed is possible.

<table>
<thead>
<tr>
<th>(\sqrt{B})</th>
<th>(\sqrt{T})</th>
<th>(I)</th>
<th>DSP48Es</th>
<th>Slices</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>16</td>
<td>6</td>
<td>1</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>10</td>
<td>1</td>
<td>34</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Mapping resource usage and latency

6. REDA Performance

First, the system model described in Sections 3 and 5 is used to show how the REDA module affects GPU memory access number \(|A|\) and variance \(\text{var}(A)\). Second, a proof of concept GPU implementation is analysed in Section 6.2 to verify the performance gain.

6.1. Analysis using a System Model

Our system model focuses on memory requests and accesses of the graphics system shown in Figure 1. It is implemented using the IEEE 1666-2005 SystemC class library. SystemC enables concurrent operations and processes to be modelled effectively. The model provides predictions for the patterns \(P, C\) and \(A\).

For the tests below a 16 processor core system is modelled with variable \(\hat{T}\) and cache size \(S\). A frame size of 512 \times 512 pixels is used with \(\hat{T}\) and \(S\) chosen as detailed below to give a representative example of a GPU system. The model can be tailored to a specific GPU architecture through changing the model parameters.

The goal is to minimise both the number of memory accesses \(|A|\) and memory access pattern variance \(\text{var}(A)\). Variance is important because DRAM performance is becoming more sensitive to memory access locality [8]. Optimisations are therefore compared against models of the original GPU architecture to justify changes in locality.

First, the model is implemented without the REDA module to highlight performance issues mentioned in Section 4 and verify the model. Second, results for including the REDA module are shown to highlight the benefits over the original system. Finally the limitations are mentioned.

<table>
<thead>
<tr>
<th>(S)</th>
<th>(\hat{T})</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KByte</td>
<td>256</td>
</tr>
<tr>
<td>32 KByte</td>
<td>256</td>
</tr>
<tr>
<td>64 KByte</td>
<td>256</td>
</tr>
<tr>
<td>128 KByte</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 2. Memory reads \(|A|\) (in KPixels) for threads \(\hat{T}\) versus cache size \(S\) for \(I, J = 16\)

The performance of the GPU is determined by the relationship of \(\hat{T}\) to cache size as shown in Table 2. Increased cache is required for a higher value of \(\hat{T}\) with an approximately linear relationship. For the results below a cache size of 16 KBytes and \(\hat{T} = 128\) are chosen. Both factors are 8 times less than predicted for the GeForce 7. This ensures that the number of threads \(\hat{T}\) does not exceed target frame size \((\hat{T} \times \hat{T})\) pixels for \(I, J\) up to 16.

Figure 4 shows the change in number of memory accesses \(|A|\) for varying window size and overlap. It is observed that for all but the 16 \times 16, memory accesses drop off significantly as overlap increases. For size 16 \times 16 number of memory reads only falls beyond 8 \times 8 overlap. The fall off for memory reads of 16 \times 16 with overlap 8 \times 8 onwards is similar to that for 8 \times 8 because the memory step \((I - O)\) between threads, explained in Section 4, is equal.

The system model with REDA is now considered. For the results that follow, only the special case \(B\) of non-overlapping windows is considered. The effects of increasing the threading factor \(\sqrt{T}\) and varying DRAM burst...
length \((B)\) are shown. This is followed by the distribution of memory accesses to show the effect on memory access locality and subsequently DRAM performance.

Table 3 shows that as the threading factor increases the memory accesses drop over the range \(\sqrt{T} = 2 \div 8\). The best case is a 60 times reduction in memory accesses.

![Figure 4. Memory reads \(|A|\) for window size versus overlap \((\hat{T} = 128, S=16\) KBytes)](image)

Table 3. Memory reads (in KPixels) for threading versus window size \((\hat{T} = 128, \sqrt{B} = 1)\)

| \(\sqrt{T}\) | \(|A|\) | \(|A|\) | \(|A|\) | \(|A|\) |
|---|---|---|---|---|
| 1 | 256 | 256 | 256 | 16385 | 16385 |
| 2 | 256 | 256 | 256 | 4096 | 4097 |
| 4 | 256 | 256 | 256 | 1024 | 1027 |
| 8 | 256 | 256 | 256 | 259 | 272 |
| 16 | 256 | 256 | 257 | 260 | 272 |

The variation of memory reads with respect to changes in \(\sqrt{B}\) and \(\sqrt{T}\) is shown in Table 4. It is seen that as \(\sqrt{B}\) increases, \(\sqrt{T}\) must be larger to reduce the number of memory reads. The table therefore shows the proportions of \(\sqrt{B}\) and \(\sqrt{T}\) that are required to overcome cache ‘thrashing’.

Although the total number of memory reads is reduced the memory access granularity (burst length) is also reduced from cache line size \((8 \times 8\) pixels\) to \(B\). This is a potential limitation of the REDA block, as it translates to a reduction in locality of memory accesses. Figure 5 shows the distribution of the original system compared to variations of \(\sqrt{B}\). A histogram is taken of the difference of consecutive memory reads (in the set \(A\)). A large peak at address difference equals one on all plots is omitted for a clearer comparison of results, and histograms are shown as a fraction of total number of memory accesses \(|A|\).

For \(\sqrt{B} = 1\) large peaks are seen away from the origin which suggests slow DRAM performance. The address mapping requires subsequent addresses to be separated in space by \(T\). This leads to poor locality. It is surprising that the distribution for \(\sqrt{B} = 2\) is more localised than the original case. This is a tradeoff between maintaining locality, with \(B = 4\), and stepping accesses to avoid cache thrashing which also reduces locality. The case where \(\sqrt{B} = 4\) also has poor overall locality which arises through the caching behaviour for the specific choice of \(I, J\) and \(\sqrt{T}\).

6.2. Enhanced GPU Performance

For an open loop test the address mapping \(R\) from Section 5.1 can be pre-computed and stored in video memory. For fixed \(I, J, O_x, O_y\) the GPU cache performance will be equivalent to a system which includes the REDA module.

This provides a proof of concept test for the mapping \(R\) and an upper bound on performance benefits. The results are an upper bound because although number of memory accesses \(|A|\) is the same, the access pattern of the proof of concept test has a greater number of sequential accesses. The experimental setup is described in Section 4.

Figure 6 shows results for varying window size \(I, J\) and threading factor \(T\), with no overlap (i.e. \(O_x, O_y = 0\)) and a memory access granularity of one \((B = 1)\).

It is seen that as threading factor \(T\) is increased the time taken per frame reduces sharply. For a small threading factor \(T\) GPU cache performance is improved significantly.

Local minima are observed, for all \(I, J\), when the threading factor is a power of two (i.e. \(\sqrt{T} = 2^n\)). These are choices of \(T\) where cache performance is superior. This justifies the choice of \(T\) as a power of two in Section 5.2.

The optimal choice of threading factor \(T\) varies with window size \(I, J\). For example, for \(I, J\) equals 16 and 8 the optimum values of \(T\) are 32 and 64 respectively.

![Figure 5. Distribution of address locality varying \(\sqrt{B}\) \((\sqrt{T}, I = 16, \hat{T} = 128, S=16\) KB\)](image)

Table 4. Memory reads (in KPixels) for threading versus block size for \(I, J = 16\) \((\hat{T} = 128)\)

<table>
<thead>
<tr>
<th>(\sqrt{T})</th>
<th>(\sqrt{B} = 1)</th>
<th>(\sqrt{B} = 2)</th>
<th>(\sqrt{B} = 4)</th>
<th>(\sqrt{B} = 8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16385</td>
<td>16385</td>
<td>16385</td>
<td>16385</td>
</tr>
<tr>
<td>2</td>
<td>4097</td>
<td>4097</td>
<td>4097</td>
<td>16385</td>
</tr>
<tr>
<td>4</td>
<td>1027</td>
<td>1027</td>
<td>4097</td>
<td>16385</td>
</tr>
<tr>
<td>8</td>
<td>272</td>
<td>260</td>
<td>257</td>
<td>16385</td>
</tr>
<tr>
<td>16</td>
<td>272</td>
<td>260</td>
<td>257</td>
<td>16385</td>
</tr>
</tbody>
</table>
The maximum performance improvement observed in Figure 6 is twenty times for a window size of $I, J = 16$.

GPU performance for increasing memory access granularity $B$ is shown in Table 5. As the granularity $B$ increases, degree to which performance can be optimised with the proposed mapping scheme reduces. For $\sqrt{B} = 2$ (current DRAM granularity) this reduction is marginal and a twelve times performance improvement is observed. This is encouraging for the proposed REDA optimisation scheme.

<table>
<thead>
<tr>
<th>$\sqrt{B}$</th>
<th>$\sqrt{T}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70.5</td>
</tr>
<tr>
<td>2</td>
<td>71.8</td>
</tr>
<tr>
<td>4</td>
<td>69.9</td>
</tr>
<tr>
<td>8</td>
<td>69.3</td>
</tr>
</tbody>
</table>

Table 5. GPU performance (ms) for threading $(T)$ versus burst length $(B)$ with $I, J = 16$

In summary it has been shown that a sixty times reduction in number of memory accesses can be achieved through using the REDA module. The effect on memory access locality has also been shown to be better than the original scenario in Figure 5. The combination of reduced number of accesses and memory access variance promises a large performance improvement from using the REDA module. An open loop test has been performed to exemplify the improvement in GPU cache performance. Up to a twenty times speedup is observed. It is concluded that the proposed architecture provides an order of magnitude performance gain over the current graphics system.

7. Conclusion

A modified graphics system is proposed which includes a reconfigurable engine for data access (REDA) to optimise memory access patterns specific to video processing. GPU performance has been analysed to identify cases of poor performance involving no or partially overlapping window functions. A REDA address mapping scheme is presented to optimise performance for this subset of memory access patterns. A sixty times reduction in number of video memory accesses is observed through analysis of a model of the graphics memory system. The results show that an order of magnitude improvement in performance is possible by using the REDA address mapping optimisation. This design, however, sacrifices memory access locality when the DRAM burst length is increased.

To generalise it has been shown that reconfigurable logic can be used to optimise the memory access performance of the graphics memory system. This utilises the data path flexibility of reconfigurable logic and the deterministic nature of video processing algorithm access patterns.

The REDA module can be applied to other memory intensive algorithms with well-defined memory access patterns which perform poorly on the GPU. Future work involves the optimisation of various access patterns using the REDA module. These include patterns with a memory access stride length which varies across a frame. Performance improvements for well-known algorithms, for example motion vector estimation, will also be analysed.

We gratefully acknowledge support from the Sony Broadcast & Professional Europe and the UK Engineering and Physical Sciences Research Council (EP/C549481/1).

References